Composing and coordinating transactional Web services

Frederic Montagut*°, Refik Molva° and Silvan Tecumseh Golega†

°Institut Eurecom
2229 Route des Cretes
06904 Sophia-Antipolis
France

*SAP Schweiz AG
Kreuzplatz 20
8008 Zürich
Switzerland

†Hasso-Plattner-Institut
Postfach 900460
D-14440 Potsdam
Germany

ABSTRACT:

Composite applications leveraging the functionalities offered by Web services are today the underpinnings of enterprise computing. However, current Web services composition systems make only use of functional requirements in the selection process of component Web services while transactional consistency is a crucial parameter of most business applications. The transactional challenges raised by the composition of Web services are twofold: integrating relaxed atomicity constraints at both design and composition time and coping with the dynamicity introduced by the service oriented computing paradigm. This chapter proposes a new process to automate the design of transactional composite Web services. This solution for Web services composition does not take into account functional requirements only but also transactional ones based on the Acceptable Termination States model. The resulting composite Web service is compliant with the consistency requirements expressed by business application designers and its execution can easily be coordinated using the coordination rules provided as an outcome of the authors' approach. An implementation of these theoretical results augmenting an OWL-S matchmaker is further detailed as a proof of concept.

KEY WORDS:
Web services, composition, termination states, transactional requirements

INTRODUCTION

Web services composition has been gaining momentum over the last years as a means of leveraging the capabilities of simple operations to offer value-added services. Complex services such as airline booking systems can be designed as the aggregation of Web services offered by different organizations. As for all cross-organizational collaborative systems, the execution of composite services requires transactional properties so that the overall consistency of data modified during the process is ensured. Yet, existing Web services composition systems appear to be limited when it comes to integrating at the composition phase, the consistency requirements defined by designers in addition to functional matchmaking. Composite Web services indeed require different transactional approaches than the ones developed for usual database systems (Elmagarmid, 1992), (Greenfield, Fekete et al. 2003). The transactional challenges raised by the composition of Web services are twofold. First, like classical workflow systems, composite services raise less stringent requirements for atomicity in that intermediate results produced by some components may be kept without rollback despite the failure to complete the overall execution of a composite service. Second, composite services are dynamic in that their components can be automatically selected at run-time based on specific requests. Existing approaches only offer means to validate transactional requirements once a composite Web service has been created (Bhiri, Perrin et al. 2005) and do not address the integration of these requirements into the composite application building process.

In this chapter, we propose a systematic procedure to automate the design of transactional composite Web services. Given an abstract representation of a process wherein instances of services are not yet assigned to component functional tasks, our solution enables the selection of Web services not only according to functional needs but also based on transactional requirements. In this approach, transactional requirements are specified by designers using the Acceptable Termination States *(ATS)* model. The resulting composite Web service is compliant with the defined consistency requirements and its execution can be easily coordinated as our algorithm also provides coordination rules that can be integrated into a transactional coordination protocol. Besides, the theoretical results developed in our approach have been implemented as a proof of concept and integrated into an OWL-S (OWL Services Coalition, 2003) functional matchmaker providing it with transactional matchmaking capabilities.
The remainder of the chapter is organized as follows. Section 2 and 3 introduce the methodology of our approach and a motivating example, respectively. In section 4, the transactional model underpinning this work is outlined. In section 5 we provide details on the termination states of a composite Web service then in section 6 we describe how transactional requirements are formed based on the properties of the termination states. The transaction-aware composition process through which transactional composite Web services are designed is detailed in section 7 while the implementation of our results in an OWL-S based framework is presented in section 8. Finally, section 9 discusses related work and section 10 presents the conclusion.

PRELIMINARY DEFINITIONS AND METHODOLOGY

Consistency is a crucial aspect of composite services execution. In order to meet consistency requirements at early stages of the service composition process, we need to consider transactional requirements a concrete parameter determining the choice of the component Web services. In this section we present a high level definition of the consistency requirements and a methodology taking into account these requirements during the composition of Web services.

Consistent composite Web services

A composite Web service W_s consists of a set of *n* Web services $W_s = (s_a)_{a\in[1,n]}$ whose execution is managed according to a workflow *W* which defines the execution order of a set of *n* tasks $W = (t_a)_{a\in[1,n]}$ performed by these services (for the sake of simplicity, we consider in our approach that a given service executes only one task). The assignment of services to tasks is performed by means of composition engines based on functional requirements. Yet, the execution of a composite service may have to meet transactional requirements aiming at the overall assurance of consistency. Our goal is to design a service assignment procedure that takes into account the transactional requirements associated with *W* in order to obtain a consistent instance W_s of *W* whose execution can be supported by a transactional protocol defined using these transactional requirements as depicted in Figure 1. We consider that each Web service component might fulfill a different set of transactional properties. For instance a service can have the capability to compensate the effects of a given operation or to re-execute the operation after failure whereas some other service does not have any of these capabilities. It is thus necessary to select the appropriate service to execute a task whose execution may be compensated if required. These transactional properties can be advertised by services in the fashion of their functional capabilities as part of their WSDL (W3C, 2002) interface or OWL-S profile. The assignment procedure based on transactional requirements follows the same strategy as the one based on functional requirements. It is a matchmaking procedure between the transactional

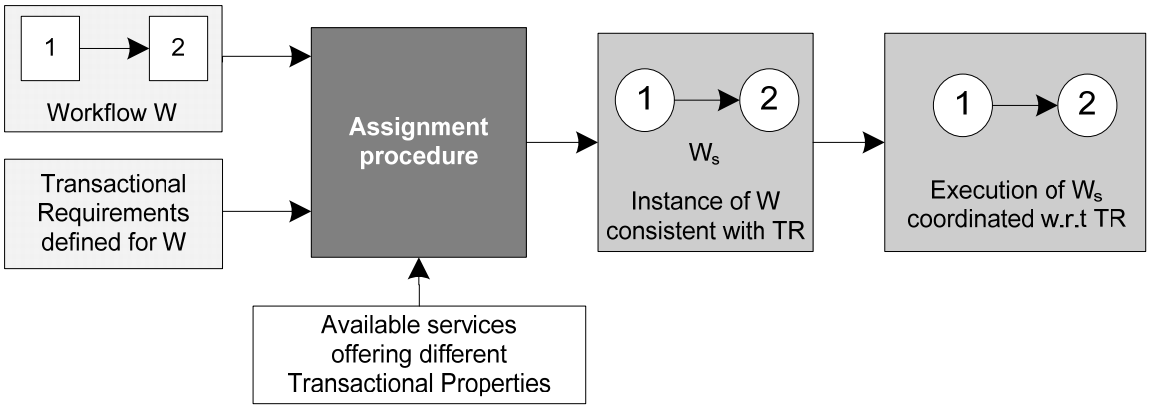


Figure 1: Principles

properties offered by services and the transactional requirements associated to each task. Once assigned, the services $(s_a)_{a\in[1,n]}$ are coordinated with respect to the transactional requirements during the process execution. The coordination protocol is indeed based on rules deduced from the transactional requirements. These rules specify the final states of execution or termination states each service has to reach so that the overall process reaches a consistent termination state. Two phase-commit (ISO, n.d.) the famous coordination protocol enforces for instance the simple rule: all tasks performed by different services have to be compensated if one of them fails. The challenges of the transactional approach are therefore twofold.

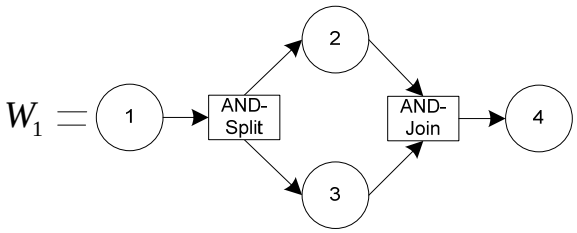
- specify a Web service assignment procedure that creates consistent instances of *W* according to defined transactional requirements
- specify the coordination rules that can be integrated into a coordination protocol managing the execution of consistent composite services

Methodology

In our approach, the candidate services for the execution of W_s are selected based on their transactional properties by means of a matchmaking procedure. We therefore need first to specify the semantic associated with the transactional properties advertised by transactional services. The matchmaking procedure is indeed based on this semantic. This semantic is also to be used in order to define a tool allowing workflow designers to specify their transactional requirements for a given workflow. Using these transactional requirements, we are able to assign services to workflow tasks based on rules which are detailed later on. Once the composite service is defined, we are able to specify the coordination rules that can be used to support the execution of the composite application according to the transactional requirements specified at the workflow design phase.

MOTIVATING EXAMPLE

In this section we introduce a motivating example that will be used throughout the chapter to illustrate the presented methodology. We consider the simple process W_1 of a manufacturing firm



$TS(W_1)$	Task 1	Task 2	Task 3	Task 4
ts_1	completed	completed	completed	completed
ts_2	completed	completed	completed	failed
ts_3	completed	compensated	completed	failed
ts_4	completed	compensated	compensated	failed
ts_5	completed	completed	compensated	failed
ts_6	compensated	compensated	compensated	failed
ts_7	compensated	completed	compensated	failed
ts_8	compensated	completed	completed	failed
ts_9	compensated	compensated	completed	failed
ts_{10}	completed	failed	completed	aborted
ts_{11}	completed	failed	compensated	aborted
ts_{12}	completed	failed	canceled	aborted
ts_{13}	compensated	failed	completed	aborted
ts_{14}	compensated	failed	compensated	aborted
ts_{15}	compensated	failed	canceled	aborted
ts_{16}	completed	completed	failed	aborted
ts_{17}	completed	compensated	failed	aborted
ts_{18}	completed	canceled	failed	aborted
ts_{19}	compensated	completed	failed	aborted
ts_{20}	compensated	compensated	failed	aborted
ts_{21}	compensated	canceled	failed	aborted
ts_{22}	failed	aborted	aborted	aborted

Figure 2: Production line process

involving four steps as depicted in Figure 2. A first service, order handling service is in charge of receiving orders from clients. These orders are then handled by the production line (step 2) and in the meantime an invoice is forwarded to a payment platform (step 3). Once the ordered item has been manufactured and the payment validated, the item is finally delivered to the client (step 4). Of course in this simple scenario, a transactional approach is required to support the process execution so that it can reach consistent outcomes as for instance the manufacturing firm would like to have the opportunity to stop the production of an item is the payment platform used by a customer is not a reliable one. On the other hand, it may no longer be required to care about canceling the production if the payment platform claims it is reliable and not prone to transaction errors. Likewise, customers may expect that their payment platform offer refunding options in case the delivery of the item they ordered is not successful.

Those possible outcomes mostly define the transactional requirements for the execution of this simple process and also specify what actions need to be taken to make sure that the final state of the process execution is deemed consistent by the involved parties. This example although simple perfectly meets our illustration needs within this chapter as it demonstrates the fact that based on the specified transactional requirements a clever selection of the business process participants has to be performed prior to the process instantiation since for instance the selection of both a payment platform that do not offer any refunding options and an unreliable delivery means may result in a disappointed customer. It should be noted that the focus of this example is not the trust relationship between the different entities and we therefore assume the trustworthiness of each of them yet we are rather interested in the transactional characteristics offered by each participant.

TRANSACTIONAL MODEL

In this section, we define the semantic specifying the transactional properties offered by services before specifying the consistency evaluation tool associated to this semantic. Our semantic model is based on the “transactional Web service description” defined in (Bhiri, Perrin et al. 2005).

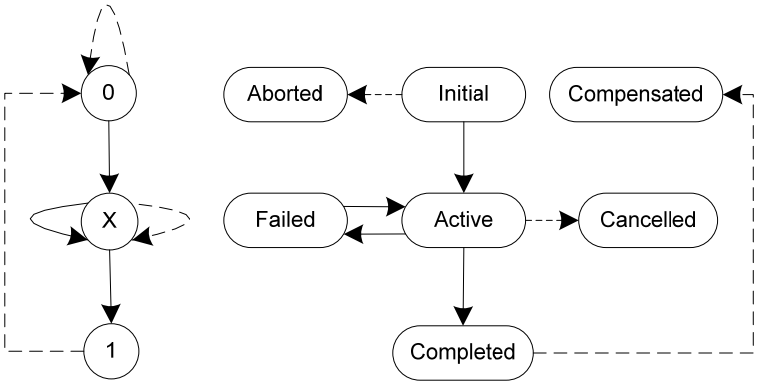


Figure 3: Service state diagram

Transactional Properties of Services

In (Bhiri, Perrin et al. 2005) a model specifying semantically the transactional properties of Web services is presented. This model is based on the classification of computational tasks made in (Mehrotra, Rastogi et al. 1992), (Schuldt, Alonso et al. 1999) which considers three different types of transactional properties. An operation and by extension a Web service executing this task can be:

- compensatable: the results produced by the task can be rolled back
- retriable: the task is sure to complete successfully after a finite number of tries
- pivot: the task is neither compensatable nor retriable

These transactional properties allow us to define four types of transactional services: retriable *(r)*, compensatable *(c)*, retriable and compensatable *(rc)* and pivot *(p)*. In order to properly understand this transactional model and the defined transactional properties, we can map the state diagram of transactional services with the state of data during the execution of computational tasks performed by these transactional services. This mapping is depicted in Figure 3. Basically, data can be in three different states: *initial* (0), *unknown* (x), *completed* (1). In the state (0), either the task execution has not yet started *initial*, the execution has been stopped, *aborted* before starting, or the execution has been properly completed and the modifications have been rolled back, *compensated*. In state (1) the task execution has been properly completed. In state (x) either the task execution is not yet finished *active*, the execution has been stopped, *canceled* before completion, or the execution has *failed*. Particularly, the states *aborted*, *compensated*, *completed*, *canceled*, and *failed* are the possible final states of execution of these tasks. Figure 4 details the transition diagram for the four types of transactional services. We distinguish within this model:

- the *inherent* termination states: *failed* and *completed* which result from the normal course of a task execution
- the *forced* termination states: *compensated*, *aborted* and *canceled* which result from a coordination message received during a coordination protocol instance and forcing a task execution to either stop or rollback

In the state diagrams of Figure 3 and Figure 4 plain and dashed lines represent the *inherent* transitions leading to *inherent* states and the *forced* transitions leading to *forced* states, respectively. In this model, the transactional properties of services are only differentiated by the states *failed* and *compensated* which indeed respectively specify the retriability and compesatability properties.

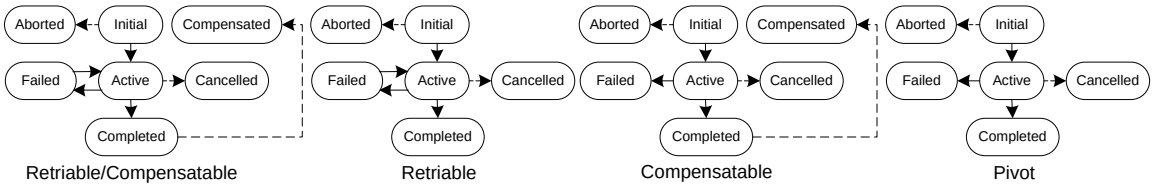


Figure 4: Transactional Properties of services

Definition 4-1: We have for a service *s*:

- *failed* is not a termination state of *s* $\Leftrightarrow$ *s* is retriable
- *compensated* is a termination state of *s* $\Leftrightarrow$ *s* is compensatable

From the state transition diagram, we can also derive some simple rules:

- The states *failed*, *completed* and *canceled* can only be reached if the service is in the state *active*.
- The state *compensated* can only be reached if the service is in the state *completed*. The state *aborted* can only be reached if the service is in the state *initial*.

Termination states

The crucial point of the transactional model specifying the transactional properties of services is the analysis of their possible termination states. The ultimate goal is indeed to be able to define consistent termination states for a workflow i.e. determining for each component service executing a workflow task which termination states it is allowed to reach.

Definition 4-2: We define the operator termination state *ts(x)* which specifies the possible termination states of the element *x*. This element *x* can be:

- a service *s* and $ts(x) \in \{aborted, canceled, failed, completed, compensated\}$
- a workflow task *t* and $ts(t) \in \{aborted, canceled, failed, completed, compensated\}$
- a workflow composed of *n* tasks $W = (t_a)_{a\in[1,n]}$ and $ts(W) = (ts(t_1), ts(t_2), ..., ts(t_n))$
- a composite service W_s of *W* composed of *n* services $W_s = (s_a)_{a\in[1,n]}$ and $ts(W_s) = (ts(s_1), ts(s_2), ..., ts(s_n))$

The operator *TS(x)* represents the finite set of all possible termination states of the element *x*, $TS(x) = (ts_k(x))_{k\in[1,j]}$. We have especially, $TS(W_s) \in TS(W)$ since the set $TS(W_s)$ represents the actual termination states that can be reached by W_s according to the transactional properties of the services assigned to workflow tasks. We also define for *x* workflow or composite service and $a \in [1,n]$.:

- $ts(x, t_a)$: the value of $ts(t_a)$ in *ts(x)*
- *tscomp(x)*: the termination state of *x* such that $\forall a \in [1,n]\ ts(x, t_a) = completed$

For the remainder of the chapter, $W = (t_a)_{a\in[1,n]}$ represents a workflow of *n* tasks and $W_s = (s_a)_{a\in[1,n]}$ a composite service of *W*.

$ATS_1(W_1)$		Task 1	Task 2	Task 3	Task 4
ats_1	ts_1	completed	completed	completed	completed
ats_2	ts_6	compensated	compensated	compensated	failed
ats_3	ts_{14}	compensated	failed	compensated	aborted
ats_4	ts_{15}	compensated	failed	canceled	aborted
ats_5	ts_{20}	compensated	compensated	failed	aborted
ats_6	ts_{21}	compensated	canceled	failed	aborted

$ATS_2(W_1)$		Task 1	Task 2	Task 3	Task 4
ats_1	ts_1	completed	completed	completed	completed
ats_2	ts_{17}	completed	compensated	failed	aborted
ats_3	ts_{11}	completed	failed	compensated	aborted
ats_4	ts_5	completed	completed	compensated	failed
ats_5	ts_{18}	completed	canceled	failed	aborted
ats_6	ts_{12}	completed	failed	canceled	aborted

Available Services		Retriable	Compensatable
Task 1	s_{11}	yes	no
	s_{12}	no	yes
	s_{13}	yes	yes
Task 2	s_{21}	yes	no
	s_{22}	no	yes
Task 3	s_{31}	yes	no
	s_{32}	no	yes
Task 4	s_{41}	no	no

Figure 5: Acceptable termination states of W_1 and available services

Transactional consistency tool

We use the Acceptable Termination States (*ATS*) (Rusinkiewicz and Sheth 1995) model as the consistency evaluation tool for our workflow. *ATS* defines the termination states a workflow is allowed to reach so that its execution is deemed consistent.

Definition 4-3: *ATS(W)* is the subset of *TS(W)* whose elements are deemed consistent by workflow designers. A consistent termination state of *W* is called an acceptable termination state $ats_k(W)$ and we note $ATS(W) = (ats_k(W))_{k\in[1,i]}$ the set of Acceptable Termination States of *W* i.e. the transactional requirements of *W*.

ATS(W) and *TS(W)* can be represented by a table which defines for each termination state the tuple of termination states reached by the workflow task as depicted in Figure 4 and Figure 5. As mentioned in the definition, the specification of the set *ATS(W)* is done at the workflow designing phase. *ATS(W)* is mainly used as a decision table for a coordination protocol so that W_s can reach an acceptable termination state knowing the termination state of a set of tasks. The role of a coordination protocol indeed consists in sending messages to component services in order to reach a consistent termination state given the current state of the workflow execution. The coordination decision, i.e. the termination state that has to be reached, made given a state of the workflow execution has to be unique; this is the main characteristic of a coordination protocol. In order to cope with this requirement, *ATS(W)* which is used as input for the coordination decision-making process has therefore to verify some properties that we detail later on.

ANALYSIS OF *TS(W)*

Since $ATS(W) \subseteq TS(W)$, *ATS(W)* inherits the characteristics of *TS(W)* and we logically need to analyze *TS(W)* first. In this section, we first make precise some basic properties of *TS(W)* derived from inherent execution rules of a workflow *W* before examining *TS(W)* from a coordination perspective.

Inherent properties of *TS(W)*

We state here some basic properties relevant to the elements of *TS(W)* and derived from the transactional model presented above. *TS(W)* is the set of all possible termination states of *W* based on the termination states model we chose for services. Yet, within a composite service execution, it is not possible to reach all the combinations represented by a *n*-tuple $(ts(t_1), ts(t_2), ..., ts(t_n))$ assuming $\forall a \in [1,n]\ ts(t_a) \in \{aborted, canceled, failed, completed, compensated\}$ **.** The first restriction is introduced by the sequential aspect of a workflow:

(P1) A task becomes *activated* ⇔ all the tasks executed beforehand according to the execution plan of *W* have reached the state *completed*

(P1) simply states that to start the execution of a workflow task, it is required to have properly *completed* all the workflow tasks required to be executed beforehand. Second, we consider in our model that only one single task can fail at a time and that the states *aborted*, *compensated* and *canceled* can only be reached by a task in a given $ts_k(W)$ if one of the services executing a task of *W* has failed. This means that the coordination protocol is allowed to force the abortion, the compensation or the cancellation only in case of failure of a service. We get *(P2)*:

(P2) if $\exists\ a,k \in [1,n] \times [1,j]$ such that $ts_k(W,t_a) \in \{aborted, canceled, compensated\} \Rightarrow$ $\exists !\, l \in [1,n]$ such that $ts_k(W,t_l) = failed$

Classification within *TS(W)*

As we explained above the unicity of the coordination decision during the execution of a coordination protocol is a major requirement. We try here to identify the elements of *TS(W)* that correspond to different coordination decisions given the same state of a workflow execution. The goal is to use this classification to specify rules to build *ATS(W)*. Using the properties *(P1)* and *(P2)*, a simple analysis of the state transition model reveals that there are two situations whereby a coordination protocol can make different coordination decisions given the state of a workflow task. Let $a,b \in [1,n]$ and assume that the task t_b has failed:

- the task t_a is in the state *completed* and either it remains in this state or it is *compensated*
- the task t_a is in the state *active* and either it is *canceled* or the coordinator lets it reach the state *completed*

From those two statements, we define the incompatibility from a coordination perspective and the flexibility notions.

Definition 5-1: Let $k,l \in [1,j]$. $ts_k(W)$ and $ts_l(W)$ are said to be incompatible from a coordination perspective $\Leftrightarrow \exists\ a,b \in [1,n]$ *such that* $ts_k(W,t_a) = completed$, $ts_k(W,t_b) = ts_l(W,t_b) = failed$ and $ts_l(W,t_a) = compensated$. Otherwise, $ts_k(W)$ and $ts_l(W)$ are said compatible from a coordination perspective. The value in $\{completed, compensated\}$ reached by a task t_a in a termination state $ts_k(W)$ whereby $ts_k(W,t_b) = failed$ is called recovery strategy of t_a against t_b in $ts_k(W)$. By extension, we can consider the recovery strategy of a set of tasks against a given task.

If two termination states are compatible, they correspond to the same recovery strategy against a given task. In fact, we have two cases for the compatibility of two termination states $ts_k(W)$ and $ts_l(W)$. Given two tasks t_a and t_b such that $ts_k(W,t_b) = ts_l(W,t_b) = failed$:

- $ts_k(W,t_a) = ts_l(W,t_a)$
- $ts_k(W,t_a) \in \{compensated, completed\}$ and $ts_l(W,t_a) \in \{aborted, canceled\}$

The second case is only possible to reach if t_a is executed in parallel with t_b. Intuitively, the failure of the service assigned to t_b occurs at different instants in $ts_k(W)$ and $ts_l(W)$.

Definition 5-2: Let $a,b \in [1,n]$. A task t_a is said to be flexible against t_b $\exists\ k \in [1, j]$ *such that* $ts_k(W,t_b) = failed$ and $ts_k(W,t_a) = canceled$. Such a termination state is said to be flexible to t_a against t_b. The set of termination states of W flexible to t_a against t_b is denoted $FTS(t_a, t_b)$.

This definition simply means that a task which is flexible against another can be canceled when the latter fails.

From these definitions, we now examine the termination states of W according to the compatibility and flexibility criteria in order to identify the termination states that follow a common strategy of coordination.

Definition 5-3: Let $a \in [1,n]$. A termination state of W $ts_k(W)$ is called generator of $t_a \Leftrightarrow$ $ts_k(W,t_a) = failed$ and $\forall b \in [1,n]$ such that t_b is executed before or in parallel with t_a, $ts_k(W,t_b) \in \{compensated, completed\}$. The set of termination states of W compatible with $ts_k(W)$ generator of t_a is denoted $CTS(ts_k(W), t_a)$.

A termination state generator of a task t_a is thus a termination state wherein t_a is in the state failed while other tasks are in the state *compensated* or *completed* if executed prior or in parallel with t_a, in the state *aborted* otherwise.

The set $CTS(ts_k(W),t_a)$ specifies all the termination states of W that follow the same recovery strategy as $ts_k(W)$ against t_a.

Definition 5-4: Let $ts_k(W) \in TS(W)$ be a generator of t_a. Coordinating an instance W_s of W in case of the failure of t_a consists in choosing the recovery strategy of each task of W against t_a and the $z_a < n$ tasks $(t_{a_i})_{i\in[1,z_a]}$ flexible to t_a whose execution is not *canceled* when t_a fails. We call coordination strategy of W_s against t_a the set:

$$CS(W_s, ts_k(W), (t_{a_i})_{i\in[1,z_a]}, t_a) = CTS(ts_k(W), t_a) - \bigcup_{i=1}^{z_a} FTS(t_{a_i}, t_a).$$

If the service s_a assigned to t_a is retriable then $CS(W_s, ts_k(W), (t_{a_i})_{i\in[1,z_a]}, t_a) = \varnothing$. W_s is said to be coordinated according to $CS(W_s, ts_k(W), (t_{a_i})_{i\in[1,z_a]}, t_a)$ if in case of the failure of t_a, W_s reaches a termination state in $CS(W_s, ts_k(W), (t_{a_i})_{i\in[1,z_a]}, t_a)$. Of course, it assumes that the transactional properties of W_s are sufficient to reach $ts_k(W)$. The coordination strategy only specifies the set of termination states that should be reached by a composite service when the latter is coordinated by means of a transactional protocol.

From these definitions, we can deduce a set of theorems.

Theorem 5-5: W_s can only be coordinated according to a unique coordination strategy at a time.
Proof: Let $a \in [1,n]$. Two termination states $ts_k(W)$ and $ts_l(W)$ both generator of t_a are incompatible.

Theorem 5-6: Let $a,k \in [1,n]\times[1,j]$ such that $ts_k(W,t_a) = failed$ but not generator of t_a. If $ts_k(W) \in TS(W_s) \Rightarrow \exists\ l \in [1,j]$ such that $ts_l(W) \in TS(W_s)$ is a generator of t_a compatible with $ts_k(W)$. This theorem states that if a composite service is able to reach a given termination state wherein a task t_a fails, it is also able to reach a termination state generator compatible with the latter.
Proof: We define $ts_l(W)$ by: $ts_l(W,t_a) = failed,\ \forall\ i \in [1,n] - \{a\}\ ts_l(W,t_i) = ts_k(W,t_i)$ if $ts_k(W,t_i) \in \{completed, compensated, aborted\},\ ts_l(W,t_i) = completed$ otherwise.

Given a task t_a the idea is to classify the elements of *TS(W)* using the sets of termination states compatible with the generators of t_a. Using this approach, we can identify the different recovery strategies and the coordination strategies associated with the failure of t_a as we decide which tasks can be *canceled*.

FORMING *ATS(W)*

Defining *ATS(W)* is deciding at design time the termination states of *W* that are consistent. *ATS(W)* is to be input to a coordination protocol in order to provide it with a set of rules which leads to a unique coordination decision in any cases. According to the definitions and properties we introduce above, we can now make explicit some rules on *ATS(W)* so that the unicity requirement of coordination decisions is respected.

Definition 6-1: Let $a,k \in [1,n]\times[1,j]$ such that $ts_k(W,t_a) = failed$ and $ts_k(W) \in ATS(W)$. *ATS(W)* is valid $\Leftrightarrow\ \exists\ !\ l \in [1,j]$ such that $ts_l(W)$ generator of t_a compatible with $ts_k(W)$ and $CTS(ts_l(W),t_a) - \bigcup_{i=1}^{z_a} FTS(t_{a_i},t_a) \subset ATS(W)$ for a set of tasks $(t_{a_i})_{i\in[1,z_a]}$ flexible to t_a.

The unicity of the termination state generator of a given task comes from the incompatibility definition and the unicity of the coordination strategy. A valid *ATS(W)* therefore contains for all $ts_k(W)$ in which a task fails a unique coordination strategy associated with this failure and the termination states contained in this coordination strategy are compatible with $ts_k(W)$. In Figure 5, an example of possible *ATS* is presented for the simple workflow W_1 of the motivating example. It just consists of selecting the termination states of the table $TS(W_1)$ that we consider consistent and respect the validity rule for the created $ATS(W_1)$. Of course for the same workflow it is possible to build different sets of acceptable termination states depending on the transactional requirements of the business application. For instance in $ATS_1(W_1)$ designers specify that the production task performed at step 2 has to be *compensated* intuitively meaning that the manufactured products have to be reprocessed whenever the delivery task fails while in $ATS_2(W_1)$ they allow these same products to remain intact.

DERIVING COMPOSITE SERVICES FROM *ATS*

In this section, we introduce a new type of service assignment procedure: the transaction-aware service assignment procedure which aims at assigning *n* services to the *n* tasks t_a in order to create an instance of *W* acceptable with respect to a valid *ATS(W)*. The goal of this procedure is to integrate within the instantiation process of workflows a systematic method ensuring the transactional consistency of the obtained composite service. We first define a validity criteria for the instance W_s of *W* with respect to *ATS(W)*, the service assignment algorithm is then detailed. Finally, we specify the coordination strategy associated to the instance created from our assignment scheme and discuss the complexity of our approach.

7.1. Acceptability of Ws with respect to *ATS(W)*

Definition 7-1: W_s is an acceptable instance of *W* with respect to *ATS(W)* $\Leftrightarrow$ $TS(W_s) \subseteq ATS(W)$.

Now we express the condition $TS(W_s) \subseteq ATS(W)$ in terms of coordination strategies. The termination state generator of t_a present in *ATS(W)* is noted $ts_{k_a}(W)$. The set of tasks whose execution is not *canceled* when t_a fails is noted $(t_{a_i})_{i\in[1,z_a]}$.

Theorem 7-2: $TS(W_s) \subseteq ATS(W) \Leftrightarrow \forall a \in [1,n]\ CS(W_s, ts_{k_a}(W), (t_{a_i})_{i\in[1,z_a]}, t_a) \subset ATS(W)$
Proof: straightforward derivation from 5-6 and 6-1.

An instance W_s of *W* is therefore an acceptable one $\Leftrightarrow$ it is coordinated according to a set of *n* coordination strategies contained in *ATS(W)*. It should be noted that if $failed \notin ATS(W, t_a)$ where $ATS(W, t_a)$ represents the acceptable termination states of the task t_a in *ATS(W)* then $CS(W_s, ts_{k_a}(W), (t_{a_i})_{i\in[1,z_a]}, t_a) = \varnothing$. From 5-6 and 7-1, we can derive the existence condition of an acceptable instance of *W* with respect to a valid *ATS(W)*.

Theorem 7-3: Let $a, k \in [1,n] \times [1,j]$ such that $ts_k(W, t_a) = failed$ and $ts_k(W) \in ATS(W)$. $\exists\ W_s$ acceptable instance of *W* with respect to *ATS(W)* such that $ts_k(W) \in TS(W_s)$ $\Leftrightarrow$ $\exists\ !\ l \in [1,j]$ such that $ts_l(W) \in TS(W_s)$ is a generator of t_a compatible with $ts_k(W)$ in *ATS(W)*.

This theorem only states that an *ATS(W)* allowing the failure of a given task can be used to coordinate a composite service also allowing the failure of the same task $\Leftrightarrow$ *ATS(W)* contains a complete coordination strategy associated to this task, i.e. it is valid.

Transaction-aware assignment procedure

In this section, we present the procedure that is used to assign services to tasks based on transactional requirements. This algorithm uses *ATS(W)* as a set of requirements during the service assignment procedure and thus identifies from a pool of available services those whose transactional properties match the transactional requirements associated to workflow tasks

defined in *ATS(W)* in terms of acceptable termination states. The assignment procedure is an iterative process, services are assigned to tasks one after the other. The assignment procedure therefore creates at each step *i* a partial instance of *W* noted W_s^i. We can define as well the set $TS(W_s^i)$ which represents the termination states of *W* that the transactional properties of the *i* services already assigned allow to reach. Intuitively the acceptable termination states refer to the degree of flexibility offered when choosing the services with respect to the different coordination strategies verified in *ATS(W)*. This degree of flexibility is influenced by two parameters:

- The list of acceptable termination states for each workflow task. This list can be determined using *ATS(W)*. This is a direct requirement which specifies the termination states allowed for each task and therefore introduces requirements on the service's transactional properties to be assigned to a given task: this service can only reach the states defined in *ATS(W)* for the considered task.
- The assignment process is iterative and therefore, as we assign new services to tasks, $TS(W_s^i)$ changes and the transactional properties required to the assignment of further services too. For instance, we are sure to no longer reach the termination states $CTS(ts_k(W),t_a)$ allowing the failure of the task t_a in *ATS(W)* when we assign a service *(r)* to t_a. In this specific case, we no longer care about the states reached by other tasks in $CTS(ts_k(W),t_a)$ and therefore there is no transactional requirements introduced for the tasks to which services have not already been assigned.

We therefore need to define first the transactional requirements for the assignment of a service after *i* steps in the assignment procedure.

Extraction of transactional requirements

From the two requirements above, we define for a task t_a :

- $ATS(W,t_a)$: Set of acceptable termination states of t_a which is derived from *ATS(W)*
- $DIS(t_a,W_s^i)$: This is the set of transactional requirements that the service assigned to t_a must meet based on the previous assignments. This set is determined based on the following reasoning:

- *(DIS1)*: the service must be compensatable $\Leftrightarrow$ $compensated \in DIS(t_a,W_s^i)$
- *(DIS2)*: the service must be retriable $\Leftrightarrow$ $failed \notin DIS(t_a,W_s^i)$

Using these two sets, we are able to compute $MIN_{TP}(s_a,t_a,W_s^i) = ATS(W,t_a) \cap DIS(t_a,W_s^i)$ which defines the transactional properties a service s_a has at least to comply with in order to be assigned to the task t_a at the *i+1* assignment step. We simply check the retriability and compensatability properties for the set $MIN_{TP}(s_a,t_a,W_s^i)$:

- $failed \notin MIN_{TP}(s_a,t_a,W_s^i) \Leftrightarrow$ s_a has to verify the retriability property
- $compensate\ d \in MIN_{TP}(s_a,t_a,W_s^i) \Leftrightarrow$ s_a has to verify the compensatability property

The set $ATS(W,t_a)$ is easily derived from *ATS(W)*. We need now to compute $DIS(t_a,W_s^i)$. We assume that we are at the *i+1* step of an assignment procedure, i.e. the current partial instance of

W is W_s^i. Computing $DIS(t_a, W_s^i)$ means determining if *(DIS₁)* and *(DIS₂) a*re true. From these two statements we can derive three properties:

1. *(DIS₁)* implies that state *compensated* can definitely be reached by t_a
2. *(DIS₂)* implies that t_a can not *fail*
3. *(DIS₂)* implies that t_a can not be *canceled*

The two first properties can be directly derived from *(DIS₁)* and *(DIS₂)*. The third one is derived from the fact that if a task can not be *canceled* when a task fails, then it has to finish its execution and reach at least the state *completed*. In this case, if a service can not be *canceled* then it can not *fail*, which is the third property. In order to verify whether 1., 2. and 3. are true, we introduce the set of theorems 7-4, 7-5 and 7-6.

Theorem 7-4: Let $a \in [1,n]$. The state *compensated* can definitely be reached by $t_a \Leftrightarrow \exists\ b \in [1,n] - \{a\}$ verifying *(7-4b)*: s_b not retriable is assigned to t_b and $\exists\ ts_k(W) \in ATS(W)$ generator of t_b such that $ts_k(W, t_a) = compensated$.

Proof: $\Leftarrow$: Since the service s_b is not retriable, it can *fail* and $ts_k(W) \in ATS(W)$ generator of t_b such that $ts_k(W, t_a) = compensated$ is in $TS(W_s)$.

$\Rightarrow$: Derived from *(P2)* and 5-6.

This theorem states that the execution of a composite service may lead a task t_a to the state *compensated* if:

- there exists a termination state part of $ATS(W)$ wherein a task t_b fails and t_a is compensated and
- t_b has been assigned to a service that is not retriable.

The two following theorems are proved similarly:

Theorem 7-5: Let $a \in [1,n]$. t_a can not *fail* $\Leftrightarrow\ \exists\ b \in [1,n] - \{a\}$ verifying *(7-5b)*: (s_b not compensatable is assigned to t_b and $\exists\ ts_k(W) \in ATS(W)$ generator of t_a such that $ts_k(W, t_b) = compensated$) or (t_b is flexible to t_a and s_b not retriable is assigned to t_b and $\forall\ ts_k(W) \in ATS(W)$ such that $ts_k(W, t_a) = failed$, $ts_k(W, t_b) \neq canceled$).

Theorem 7-6: Let $a, b \in [1,n]$ such that t_a is flexible to t_b. t_a is not *canceled* when t_b *fails* $\Leftrightarrow$ *(7-6b)*: s_b not retriable is assigned to t_b and $\forall\ ts_k(W) \in ATS(W)$ such that $ts_k(W, t_b) = failed$, $ts_k(W, t_a) \neq canceled$.

According to the theorems 7-4, 7-5 and 7-6, in order to compute $DIS(t_a, W_s^i)$, we have to compare t_a with each of the *i* tasks $t_b \in W - \{t_a\}$ to which a service s_b has been already assigned. This is an iterative procedure and at the initialization phase, since no task has been yet compared to t_a, s_a can be of type *(p)*: $DIS(t_a, W_s^i) = \{failed\}$.

1. if t_b verifies *(7-4b)* $\Rightarrow\ compensated \in DIS(t_a, W_s^i)$
2. if t_b verifies *(7-5b)* $\Rightarrow\ failed \notin DIS(t_a, W_s^i)$

3. if t_b is flexible to t_a and verifies *(7-6b)* $\Rightarrow$ $failed \notin DIS(t_a, W_s^i)$

The verification stops if $failed \notin DIS(t_a, W_s^i)$ and $compensated \in DIS(t_a, W_s^i)$. With $MIN_{TP}(s_a, t_a, W_s^i)$, we are able to select the appropriate service to be assigned to a given task according to transactional requirements.

Service assignment process

Services are assigned to each workflow task based on an iterative process. Depending on the transactional requirements and the transactional properties of the services available for each task, different scenarios can occur:

(i) services of type *(rc)* are available for the task. It is not necessary to compute transactional requirements as such services match all transactional requirements.

(ii) only one service is available for the task. We need to compute the transactional requirements associated to the task and either the only available service is sufficient or there is no solution.

(iii) services of types *(r)* and *(c)* but none of type *(rc)* are available for the task. We need to compute the transactional requirements associated to the task and we have three cases. First, (retriability and compensatability) is required in which case there is no solution. Second, retriability (resp. compensatability) is required and we assign a service of type *(r)* (resp. *(c)*) to the task. Third, there is no requirement.

The idea is therefore to assign first services to the tasks verifying *(i)* and *(ii)* since there is no flexibility in the choice of the service. Tasks verifying *(iii)* are finally analyzed. Based on the transactional requirements raised by the remaining tasks, we first assign services to tasks with a non-empty transactional requirement. We then handle the assignment for tasks with an empty transactional requirement. Note that the transactional requirements of all the tasks to which services are not yet assigned are also affected (updated) as a result of the current service assignment. If no task has transactional requirements then we assign the services of type *(r)* to assure the completion of the remaining tasks' execution.

Theorem 7-7: The service assignment procedure creates an instance of *W* that is acceptable with respect to a valid *ATS(W)*.

Proof: Let W_s be an instance of *W* resulting from the service assignment procedure and a service s_a assigned to a task t_a in W_s. The definition 7-1 has to be verified and we therefore consider *(A)* and *(B)* (keeping the notations of theorem 7-2):

(A) $\forall\ a \in [1,n],\ failed \in ATS(W, t_a) \Rightarrow CS(W_s, ts_{k_a}(W), (t_{a_i})_{i\in[1,z_a]}, t_a) \subset ATS(W)$

(B) $\forall\ a \in [1,n],\ failed \notin ATS(W, t_a) \Rightarrow CS(W_s, ts_{k_a}(W), (t_{a_i})_{i\in[1,z_a]}, t_a) \subset ATS(W)$

(A): We suppose that $failed \in ATS(W, t_a)$ then we have two possibilities:

- s_a is retriable and $CS(W_s, ts_{k_a}(W), (t_{a_i})_{i\in[1,z_a]}, t_a) = \varnothing \subset ATS(W)$.
- s_a can *fail* and with 1., 2. and 3. we get $ts_{k_a}(W) \in TS(W_s)$ and therefore $CS(W_s, ts_{k_a}(W), (t_{a_i})_{i\in[1,z_a]}, t_a) \subset ATS(W)$ since *ATS(W)* is valid.

(B): We suppose that $failed \notin ATS(W, t_a)$ then $failed \notin MIN_{TP}(s_a, t_a, W_s^i)$ and s_a is retriable. Therefore, $CS(W_s, ts_{k_a}(W), (t_{a_i})_{i\in[1,z_a]}, t_a) = \varnothing \subset ATS(W)$.

Finally, we get $CS(W_s, ts_{k_a}(W), (t_{a_i})_{i\in[1,z_a]}, t_a) \subset ATS(W)$ and W_s is an acceptable instance of *W* with respect to *ATS(W)*.

Coordination of W_s

Now, using *(A)* and *(B)* defined in the proof of 7-7 and keeping the same notations, we are able to specify the coordination strategy of W_s against each workflow task. We get indeed the following theorem.

Theorem 7-8: Let W_s be an acceptable instance of *W* with respect to *ATS(W)*. We note $(t_{a_i})_{i\in[1,n_r]}$ the set of tasks to which no retriable services have been assigned. We get:

$$TS(W_s) = \{tscomp(W_s)\} \cup \bigcup_{i=1}^{n_r} \left(CTS(ts_{k_{a_i}}(W), t_{a_i}) - \bigcup_{j=1}^{z_a} FTS(t_{a_{i_j}}, t_{a_i}) \right)$$

Having computed $TS(W_s)$, we obtain the list of the possible termination states that can be reached by the instance W_s and thus that defines the coordination rules associated with the execution of W_s. $TS(W_s)$ is indeed derived from *ATS(W)* which contains for all tasks at most a single coordination strategy as specified in 6-1. As a result, whenever the failure of a task t_a is detected, a transactional protocol in charge of coordinating an instance W_s resulting from our approach reacts as follows. The coordination strategy $CS(W_s, ts_{k_a}(W), (t_{a_i})_{i\in[1,z_a]}, t_a)$ corresponding to t_a is identified and a unique termination state belonging to $CS(W_s, ts_{k_a}(W), (t_{a_i})_{i\in[1,z_a]}, t_a)$ can be reached given the current state of the workflow execution.

Discussion

The operations that are relevant from the complexity point of view are twofold: the definition of transactional requirements by means of the acceptable termination states model and the execution of the transaction-aware service assignment procedure.

One can argue that building an *ATS* table specifying the transactional requirements of a business process *W* consists of computing the whole *TS(W)* table, yet this is not the case. Building a *ATS(W)* set in fact only requires for designers to identify the tasks of *W* that they allow to fail as part of the process execution and to select the termination state generator associated with each of those tasks that meet their requirements in terms of failure atomicity. Once this phase is complete, designers only need to select the tasks whose execution can be canceled when the former tasks may fail and complete the associated coordination strategy.

The second aspect concerns the complexity of the transaction aware assignment procedure that we presented in section 6 and 7.

Theorem 7-9: Let $W = (t_a)_{a\in[1,n]}$ a workflow. The complexity of the transaction-aware assignment procedure is $O(n^3)$**.**

Proof: We can show that the number of operations necessary to compute the step *i* of the assignment procedure for a task t_a is bounded by $4 \times n \times i$**.** Computing the step *i* indeed consists

of verifying the theorems 7-4, 7-5 and 7-6 and determining $ATS(W,t_a)$. On the one hand, performing the operations part of theorems 7-4 (one comparison), 7-5 (two comparisons) and 7-6 (one comparison) requires at most 4 comparisons. On the other hand, building $ATS(W,t_a)$ requires at most *n* operations (there is at most *n* generators in a *ATS(W)* set). Therefore, we can derive that the number of operations that needs to be performed in order to compute the *n* steps of the assignment procedure for a workflow composed of *n* tasks is bounded by $4\times n\times\left(\sum_{j=1}^{n} j\right)$ which is equivalent to n^3 as $n\to\infty$.

Example

Back to our motivating example, we consider the workflow W_1 of Figure 2. Designers have defined $ATS_2(W_1)$ as the transactional requirements for the considered business application and the set of available services for each task of W_1 is specified in Figure 5. The goal is to assign services to workflow tasks so that the instance of W_1 is valid with respect to $ATS_2(W_1)$ and we apply the assignment procedure presented in section 7.2. We first start to assign the services of type *(rc)* for which it is not necessary to compute any transactional requirements. s_{13} which is available for task 1 is therefore assigned without any computation. We then consider the tasks for which only one service is available. This is the case for task 4 for which only one service of type *(p)* is available. We therefore verify whether s_{41} can be assigned to task 4. We compute $MIN_{TP}(s_a,t_4,W_{1s}^1)=ATS_2(W_1,t_4)\cap DIS(t_4,W_{1s}^1)$. $ATS_2(W_1,t_4)=\{completed,\ failed\}$ and $DIS(t_4,W_{1s}^1)=\{failed\}$ as s_{13} the only service already assigned is of type *(rc)* and the theorems 7-4, 7-5 and 7-6 are not verified, none the conditions required within these theorems are indeed verified by the service s_{13}. Thus $MIN_{TP}(s_a,t_4,W_{1s}^1)=\{failed\}$ and s_{41} can be assigned to task 4 as it matches the transactional requirements. Now we compute the transactional requirements of task 2 for which services of type *(r)* and *(c)* are available and we get $MIN_{TP}(s_a,t_2,W_{1s}^2)=\{failed\}$. As described in the assignment procedure we do not assign any service to this task as it does not introduce at this step of the procedure any transactional requirements to make a decision on the candidate service to choose. We therefore compute the transactional requirements of task 3 and we get $MIN_{TP}(s_a,t_3,W_{1s}^2)=\{failed,compensated\}$ as theorem 7-4 is verified with the service s_{41} that is indeed not retriable. The service s_{32} which is of type *(c)* can thus be assigned to task 3 as it matches the computed transactional requirements. We come back now to task 2 and compute the transactional requirements once again and we get $MIN_{TP}(s_a,t_2,W_{1s}^3)=\{failed,compensated\}$ as theorem 7-4 is now verified with the service s_{32} which is indeed not retriable. It should be noted that at this step, the transactional requirements associated to task 2 have been modified because of the assignment of the service s_{32} to task 3. As the device s_{22} matches the transactional requirements it can be assigned to the task.

IMPLEMENTATION

To implement the above presented work we augmented an existing functional OWL-S matchmaker (Tang, Liebetruth et al. 2003), with transactional matchmaking capabilities. In order to achieve our goal, the matchmaking procedure has been split into two phases. First, the functional

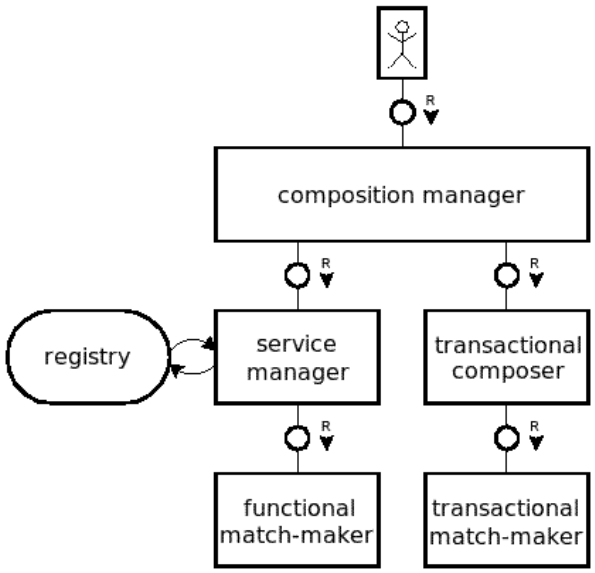


Figure 6: Transactional Web services composition system

matchmaking based on OWL-S semantic matching is performed in order to identify subsets of the available services that meet the functional requirements for each workflow task. Second, the implementation of the transaction-aware service assignment procedure is run against the selected sets of services in order to build an acceptable instance fulfilling defined transactional requirements.

The structure of the matchmaker consists of several components whose dependencies are displayed in Figure 6. The composition manager manages the process of matchmaking and provides a Java API that can be invoked to start the composition. It gets as input an abstract process description specifying the functional requirements for the candidate services and a table of acceptable termination states. The registry stores OWL-S profiles of services that are available. Those OWL-S profiles have been augmented with the transactional properties offered by services. This has been done by adding to the non-functional information of the OWL-S profiles a new element called *transactionalproperties* that specifies two Booleans attributes *retriable* and *compensatable* as shown in the sample listing below:

```
<tp:transactionalproperties retriable="true" compensatable="true"/>
```

In the first phase of the composition procedure, the service manager is invoked with a set of OWL-S profiles that specify the functional requirements for each workflow task. The service manager gets access to the registry where all published profiles are available and to the functional matchmaker provided by (Tang, Liebetruth et al. 2003) and that is used to match the available profiles against the functional requirements specified in the workflow. For each workflow task, the service manager then returns a set of functionally matching profiles along with their transactional properties. The composition manager then initiates the second phase, passing these sets along with the process description, and the table of acceptable termination states to the transactional composer. The transactional composer starts the transaction-aware service

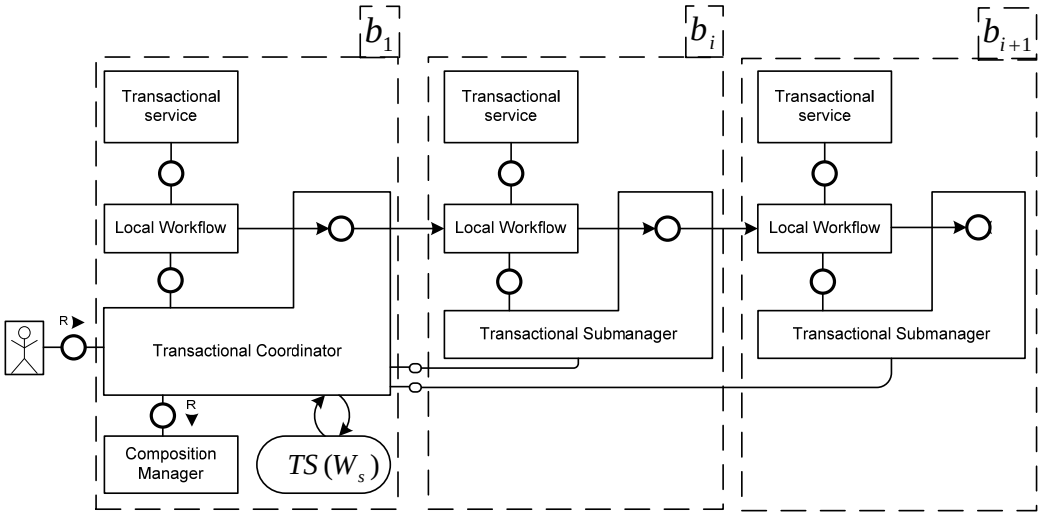


Figure 7: Transactional Architecture

assignment procedure using the transactional matchmaker by classifying first those sets into five groups:

- sets including only services of type *(p)*
- sets including only services of type *(r)*
- sets including only services of type *(c)*
- sets including services of types *(r)* and *(c)*
- sets including services of type *(rc)*

Once those sets are formed, the iterative transactional composition process is performed as specified above based on the table of acceptable termination states defined for the process. Of course depending on the set of available services and the specified acceptable termination states, the algorithm execution may end without yielding a solution. The implementation work we have performed reveals that the execution overhead introduced by our transaction-aware assignment procedure within the complete service composition procedure is in fact negligible with respect to the time required to parse OWL-S documents and execute the functional match-making procedure.

Coordination

In this section we suggest a coordination framework for transactional composite applications, based on Web services technologies. To illustrate the latter, we introduce a scenario featuring the transactional coordination of a cross-organizational composite application that is built based on our transaction-aware assignment procedure. To that respect, the business partners involved in the composite application share their services and communicate through local workflow engines that help them manage the overall collaboration in a distributed manner. The system architecture is depicted in Figure 7. In order to support the execution of cross-organizational composite applications, we designed in the fashion of the WS-Coordination initiative (Langworthy, 2005) a transactional stack composed of the following components:

- **Transactional coordinator**: this component is supported by the composite application initiator. On the one hand it implements the transaction-aware business partner assignment procedure as part of the composition manager module and on the other hand

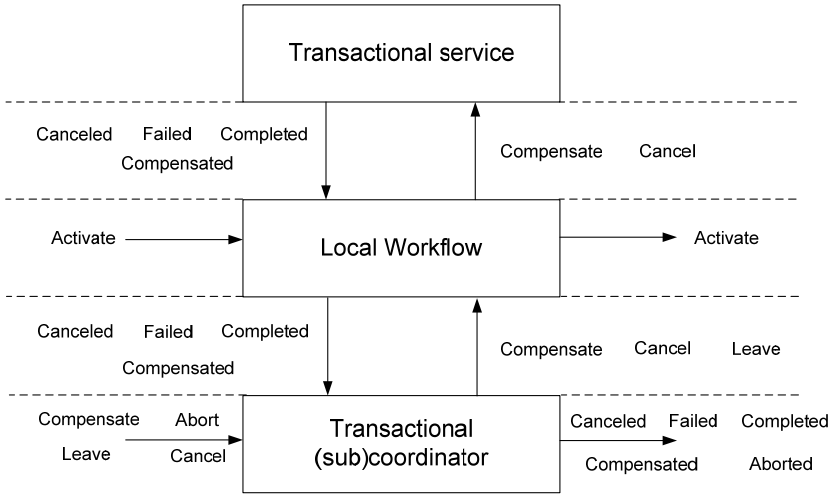


Figure 8: Infrastructure internal communications

it is in charge of assuring the coordinator role relying on the set *TS(Ws)* outcome of the assignment procedure.

- **Transactional submanager**: this component is deployed on the other partners and is in charge of forwarding coordination messages from the local workflow to the coordinator and conversely.

In the infrastructure that is deployed on each business partner to implement the coordination framework presented in this section, the transactional coordinator plays the role of interface between the business process and the other business partners when it comes to managing the notification messages exchanged based on the coordination rules extracted from *TS(Ws)*. Some of these messages received by the transactional coordinator should be forwarded to the local business process to take appropriate actions while some others are only relevant to the local transactional (sub)coordinator. The business process may also require to issue a notification to its local transactional (sub)coordinator when a failure occurs. The messages exchanged between these three layers are derived from the state model depicted in Figure 3. The infrastructure deployed on a given business partner basically consists of three layers:

- **The transactional service layer** representing the business partner's available operations,
- **The local workflow layer** corresponding to the local workflow engine,
- **The coordination layer** implementing the local (sub)coordinator module.

The message exchanges that can take place on a given business partner between these three layer are depicted in Figure 8 and specified as follows.

- **Activate**: The activate message is basically issued by the local workflow engine to the local workflow engine of the next business partner involved in the workflow. In fact this message instantiates the process execution on the business partner side.
- **Compensate, Cancel**: The compensate and cancel messages are received at the coordination layer and forwarded to the local workflow layer that forwards them in a second time to the transactional service layer to perform to corresponding functions i.e. compensation or cancellation of an operation.
- **Compensated, Cancelled, Completed**: These messages simply notify that the corresponding events have occurred: compensation, cancellation, or completion of an

operation. Issued at the transactional service layer, they are forwarded to the coordination layer in order to be dispatched to the composite application coordinator.

- **Failed**: Issued at the transactional service layer, the failed message is forwarded to the coordination layer in order to be dispatched to the composite application coordinator. If the operation performed at the transactional service layer is retriable, no failed message is forwarded to the local workflow layer as we consider that the retry primitive is inherent to any retriable operation.
- **Abort, Aborted**: The abortion message is received at the coordination layer and acknowledged with an aborted message. Upon receipt of this message, the business simply leaves the composite application execution; no message is forwarded to the other layers since the local workflow has not yet been instantiated.

RELATED WORK

Transactional consistency of workflows and database systems has been an active research topic over the last 15 years yet it is still an open issue in the area of Web services (Curbera, Khalaf et al. 2003), (Gudgin, 2004), (Little, 2003) and especially composite Web services. Composite Web services indeed introduce new requirements for transactional systems such as dynamicity, semantic description and relaxed atomicity. Existing transactional models for advanced applications (Elmagarmid, 1992) lack flexibility to integrate these requirements (Alonso, Agrawal et al. 1996) as for instance they are not designed to support the execution of dynamically generated composite services. Our solution allows the specification of transactional requirements supporting relaxed atomicity for an abstract workflow specification and the selection of a semantically described service set meeting the transactional requirements defined at the workflow design stage.

Our work is based on (Bhiri, Perrin et al. 2005) which presents the first approach specifying relaxed atomicity requirements for composite Web services based on the ATS tool and a transactional semantic. Despite a solid contribution, this work appears to be limited if we consider the possible integration into automatic Web services composition systems. It indeed only details transactional rules to validate a given composite service with respect to defined transactional requirements. In this approach, transactional requirements do not play any role in the component services selection process which may result in several attempts for designers to determine a valid composition of services. On the contrary, our solution provides a systematic procedure enabling the automatic design of transactional composite Web services. Besides, our contribution also defines the mathematical foundations to specify valid *ATS* for workflows based on the concept of coordination strategy.

Finally, our solution can be used to augment recent standardization efforts in the area of transactional coordination of Web services (Abbott, 2005), (Langworthy, 2005). Our approach indeed provides adaptive coordination specifications based on the transactional properties of the component services instantiating a given workflow. Existing Web services coordination specifications (Langworthy, 2005) are indeed not flexible enough as they do not neither allow workflow designers to specify their transactional requirements nor take into account the transactional properties offered by Web services.

CONCLUSION

We presented a systematic procedure to automate the design of transactional composite Web services. Our solution enables the selection of component Web services not only according to

functional requirements but also to transactional ones. Transactional requirements are defined by designers and serve as an input to define both reliable composite Web services and coordination protocols used to ensure the consistency of their execution. On the one hand this service assignment approach can be used to augment existing Web services composition systems (Agarwal, Dasgupta, et al. 2005) as it can be fully integrated in existing functional matchmaking procedures. On the other hand, our approach defines adaptive coordination rules that can be deployed on Web services coordination specifications (Langworthy, 2005) in order to increase their flexibility. Besides, the theoretical results presented in this chapter have been integrated into an OWL-S matchmaker as a proof of concept.

ACKNOWLEGMENT

This work has been partially sponsored by EU IST Directorate General as a part of FP6 IST projects MOSQUITO and R4eGov and by SAP Labs France S.A.S.

REFERENCES

Abbott, M. (2005). *Business transaction protocol*.

Agarwal, V., Dasgupta, K., Karnik, N., Kumar, A., Kundu, A., Mittal, S., & Srivastava, B. (2005), A service creation environment based on end to end composition of web services. In *Proceedings of the WWW conference* (pp. 128-137). Chiba, Japan,.

Alonso, G., Agrawal, D., Abbadi, A. E., Kamath, M., Gnthr, R., & Mohan, C. (1996). Advanced transaction models in workflow contexts. In *Proceedings of the 12th International Conference on Data Engineering*, New Orleans (pp. 574-581).

Bhiri, S., Perrin, O., & Godart, C. (2005). Ensuring required failure atomicity of composite web services, *Proceedings of the WWW conference* (pp. 138-147). Chiba, Japan.

Curbera, F., Khalaf, R., Mukhi, N., Tai, S., & Weerawarana, S. (2003). The next step in web services. *Communications of the ACM, 46*(10), 29-34.

Elmagarmid, A. K. (1992). *Database Transaction Models for Advanced Applications*. Morgan Kaufmann.

Greenfield, P., Fekete, A., Jang, J., & Kuo, D. (2003), Compensation is not enough. In *Proceedings of the 7th International Enterprise Distributed Object Computing Conference (EDOC'03)*, 232. Brisbane, Australia.

Gudgin, M. (2004), Secure, reliable, transacted; innovation in web services architecture,. In*Proceedings of the ACM International Conference on Management of Data* (pp. 879-880).

Langworthy, D. (2005). *WS-AtomicTransaction.*

Langworthy, D. (2005). *WS-BusinessActivity.*

Langworthy, D. (2005). *WS-Coordination.*

Little, M. (2003), Transactions and web services. *Communications of the, 46*(10), 49–54.

Mehrotra, S., Rastogi, R., Silberschatz, A., & Korth, H. (1992), A transaction model for multidatabase systems. In *Proceedings of the 12th IEEE International Conference on Distributed Computing Systems (ICDCS92)* (pp. 56-63). Yokohama, Japan.

OWL Services Coalition. (2003). *OWL-S: Semantic Markup for Web Services.*

Rusinkiewicz, M., & Sheth, A. (1995). Specification and execution of transactional workflows. *Modern database systems: the object model, interoperability, and beyond* (pp. 592–620).

Schuldt, H., Alonso, G., & Schek, H. (1999) Concurrency control and recovery in transactional process management. In *Proceedings of the Conference on Principles of Database Systems* (pp. 316-326). Philadelphia, Pennsylvania.

Tang, S. Liebetruth, C., & Jaeger, M. C. (2003) The OWL-S matcher software. Retrieved from http://flp.cs.tu-berlin.de/

Thatte, S. (2003). *Business Process Execution Language for Web Services Version 1.1 (BPEL).*

W3C. (2002).*Web Services Description Language (WSDL).*

ISO. (n.d.). *Open System Interconnection- Distributed Transaction Processing (OSI-TP) Model, ISO IS 100261.*